DOCUMENT RESUME

ED 088 474 IR 000 307

AUTHOR	Fitzhugh, Robert J.
TITLE	A General-Purpose Time-Sharing System For a Small- Or Medium-Scale Computer.
INSTITUTION	Pittsburgh Univ., Pa. Learning Research and Development Center.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	PU-LRDC-1973-23
PUB DATE	Nov 73
NOTE	23p.
EDRS PRICE	MF-$0.75 HC-$1.50
DESCRIPTORS	*Computer Oriented Programs; *Computers; *Computer Science; Program Descriptions; *Time Sharing
IDENTIFIERS	Batch Processing; ETSS; *Experimental Time Sharing System; Memory Management; Task Scheduling

ABSTRACT

An overview is provided of the Experimental Time-Sharing System (ETSS). Few manufacturers of small-scale and medium-scale computers offer operating systems capable of supporting both terminal-oriented time-sharing and multiprogramed batch processing. ETSS was developed to demonstrate that a powerful, general purpose operating system of this type is feasible on a smaller computer and that it can provide highly flexible and cost-effective computer services. Developed for an educational application, the system can be tailored to meet a wide range of application requirements and can support a mix of terminal-oriented, time-sharing, fast response process control and one or more batch streams. Described in this paper are the organization of the operating system and its major components, with detailed attention being given to task scheduling and memory management. (Author)

A GENERAL-PURPOSE TIME-SHARING SYSTEM FOR A SMALL- OR MEDIUM-SCALE COMPUTER

Robert J. Fitzhugh

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

Learning Research and Development Center

University of Pittsburgh

November 1973

The research reported herein was supported by a grant from the National Science Foundation (NSF-GJ-540X) and by the Learning Research and Development Center, supported in part by funds from the National Institute of Education, United States Department of Health, Education, and Welfare. The opinions expressed do not necessarily reflect the position or policy of the sponsoring agencies and no official endorsement should be inferred.

Abstract

Few manufacturers of small- and medium-scale computers offer operating systems capable of supporting both terminal-oriented time-sharing and multiprogrammed batch. The Experimental Time-Sharing System (ETSS) was developed as a demonstration that a powerful, general-purpose operating system of this type is feasible on a smaller computer and that it can provide highly flexible and cost-effective computer services. Developed for an educational application, the system can be tailored or 'tuned' to meet a wide range of application requirements and can support a mix of terminal-oriented time-sharing, fast-response process control and one or more batch streams.

This paper provides an overview of ETSS and describes the organization of the operating system and its major components. Although much has to be abbreviated in the interest of space, task scheduling and memory management are described in some detail.

A GENERAL-PURPOSE TIME-SHARING SYSTEM FOR A SMALL- OR MEDIUM-SCALE COMPUTER

Robert J. Fitzhugh

Learning Research and Development Center

University of Pittsburgh

Introduction

Few manufacturers of small- and medium-scale computers offer operating systems capable of supporting both terminal-oriented time-sharing and multiprogrammed batch. The Experimental Time-Sharing System (ETSS) was developed as a demonstration that a powerful, general-purpose operating system of this type is feasible on a smaller computer and that it can provide highly flexible and cost-effective computer services. Developed for an educational application, the system can be tailored or 'tuned' to meet a wide range of application requirements and can support a mix of terminal-oriented time-sharing, fast-response process control and one or more batch streams.

Based on a DEC PDP-15 computer with a fast, swapping drum, the system was designed for machine independence so that a later conversion to a multiprocessing environment or to another computer would be possible. In its current implementation, over 95 percent of the operating system consists of pure procedures which have and require no knowledge of the processor's input/output structure, memory protection structure, or interrupt structure.

This paper provides an overview of ETSS and describes the organization of the operating system and its major components. Although much

must be omitted in the interest of space, task scheduling, and memory management is described in some detail.

System Organization

The ETSS operating system consists of five major procedures, a collection of peripheral device control subprocedures, a body of common subroutines and a variety of tables and context blocks, some permanent and some dynamically created and destroyed through time. These operating system components reside in a portion of main memory called 'syspace' for system space. User programs execute in the remaining portion of memory called 'uspace' for user space. When required, portions of uspace may be transferred to and from an auxiliary swapping device.

Major Procedures. Although the five procedures share a body of common subroutines, each procedure is independent of the others and is responsible for a major system function. Figure 1 shows the five procedures and the lines of communication between them. The EXECUTIVE procedure is the heart of the operating system and is responsible for the allocation and control of all memory and computational resources. Any procedure can request the EXECUTIVE to create a 'task' (or job) with a specified set of characteristics and schedule it for execution. During its lifetime, the task is under the exclusive control of the EXECUTIVE although the procedure that requested its creation can also request the EXECUTIVE to suspend or destroy the task at any time.

The requests to create and destroy tasks most commonly come from the BATCH and MONITOR procedures although the INPUT/OUTPUT

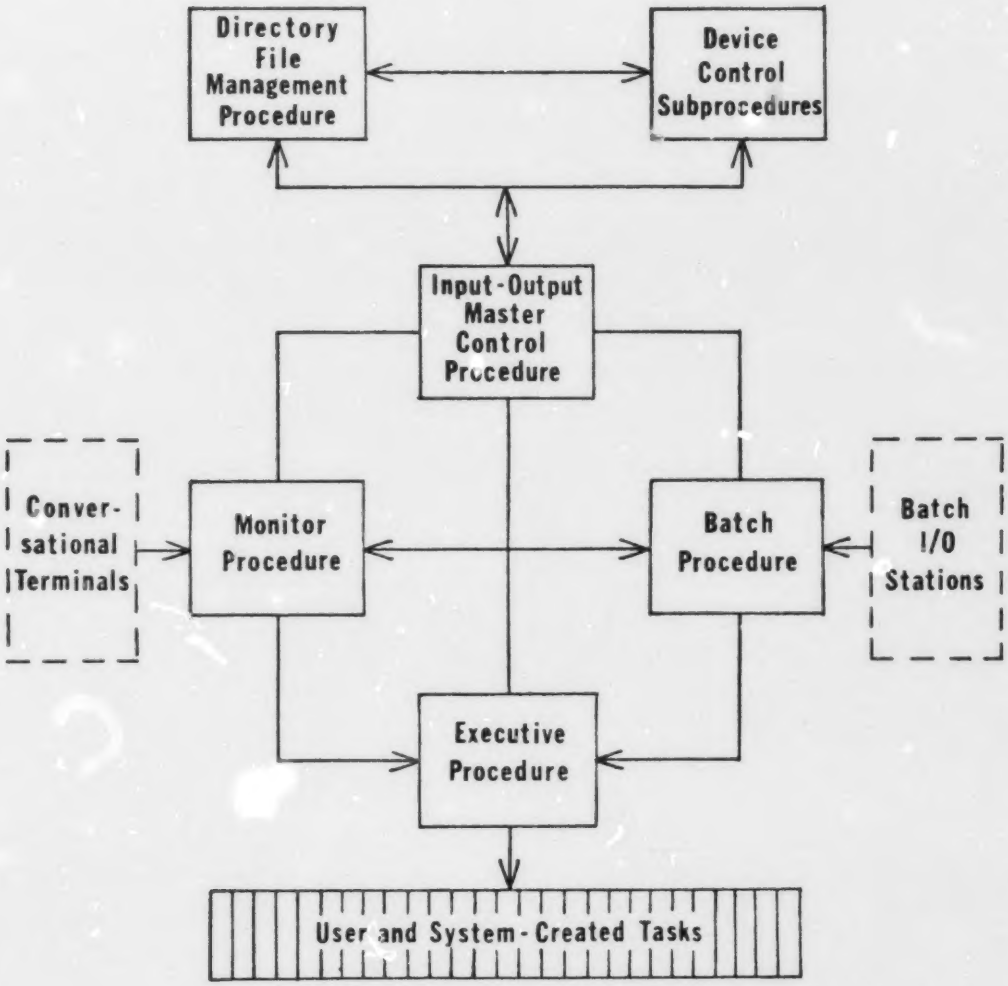

Figure 1. Major System Procedures and Their Lines of Communication

MASTER CONTROL (IOMC) procedure does request the creation of input/output spooling tasks. The BATCH and MONITOR procedures each function as a software interface between the EXECUTIVE procedure and the human users of the system. The MONITOR procedure supports conversational time-sharing and provides the user at a terminal with a MONITOR Command Language (MCL) through which the user is able to enter and exit the system, initiate tasks, acquire and return system resources, display and modify dataset directories, request information on system status and performance and invoke a wide variety of language processors and utilities. The BATCH procedure maintains one or more batch processing streams and interacts with batch input/output stations. A BATCH Command Language (BCL) is provided with which tasks to be run are specified. Although the MONITOR and BATCH procedures may create tasks with different characteristics, the tasks are indistinguishable to the EXECUTIVE and are able to share all system resources including a common file structure.

The INPUT/OUTPUT MASTER CONTROL (IOMC) procedure is responsible for the allocation and control of all input/output resources including input/output channels, space on auxiliary storage devices, and access to system peripherals. By masking hardware idiosyncrasies, the IOMC procedure enables user programs and other system procedures to interface with a virtual input/output structure in which programs reference 'files' assigned to any appropriate device or dataset. The IOMC procedure serves as a translator converting these file-oriented input/output requests into requests to actual physical devices or datasets.

During the processing of a file request, the IOMC procedure determines if the request refers to a 'directory' or to a 'non-directory'

device. Directory devices contain data as well as directories with the names and the locations of the permanent entries or datasets stored on the device. A disk unit might be used as a directory device, and a line printer would be a typical non-directory device. If the file request refers to a non-directory device, the IOMC procedure interacts directly with the appropriate device control subprocedure. All requests referencing directory devices are sent by the IOMC procedure to the DIRECTORY FILE MANAGEMENT (DFM) procedure.

The DFM procedure permits the programmer to treat a directory device as a file-oriented medium without any concern for the device's physical properties or addressing structure. Data written to a directory device can be left in a temporary state or it can be cataloged as a dataset. Datasets may be cataloged in a user's private directory, in a directory available to all called the 'library,' or in the directory of another user if permitted. Protection keys are provided to restrict dataset access when required. During the processing of a request, DFM interacts with the one or more device control subprocedures responsible for the control of the actual physical devices involved.

Inter-Procedure Communication. The five procedures, MONITOR, BATCH, EXECUTIVE, IOMC, and DFM, are functionally independent and communicate with each other in a standard and well-defined fashion. Each procedure is assigned to a hardware priority interrupt channel on the machine, and each can be viewed as a large, interrupt subroutine. This last point is important and must be understood to grasp the structure of ETSS. It should be mentioned that a unique interrupt channel is not required for each procedure although three of the five procedures do have a unique channel in the current implementation. The procedures are ordered by priority with procedures such as the EXECUTIVE, which

is responsible for task scheduling, operating at a higher priority level than the MONITOR procedure.

Each procedure has a 'request list' and a 'done list ' each a bi-directional, circular-linked list as are all ETSS lists. A procedure desiring services from another creates a 'message packet,' inserts the packet in the other procedure's request list and triggers a hardware interrupt on that procedure's interrupt channel. The message packet defines the request and consists of a variable number of parameters in a standard format. The procedure stimulated by the interrupt first scans its done list and then its request list, processing in turn each message packet that it encounters. At the completion of a request, the procedure performing the service inserts the packet in the done list of the original requesting procedure and triggers the appropriate hardware interrupt.

Common subroutines are available to handle message packet creation, deletion, and transmission between procedures. These subroutines access several tables containing implementation specific information on each procedure. This level of detail is transparent to the procedures themselves, and a procedure need only specify the destination procedure for packet transmission to occur. The procedure performing the service simply requests that the packet be returned leaving the responsibility of determining the original sender to the common subroutine.

System Tables and Context Blocks. Data defining the system and describing its current status are stored in a variety of internal tables and context blocks. Some of these data structures are permanently resident within the operating system and others are dynamically created

and destroyed in response to changing conditions. The context block structure is particularly important and is a central feature of the overall system design about which the remainder of the operating system is organized. Any oversights or inflexibilities in this area would be difficult or impossible to remedy later.

The operating system maintains four types of context blocks. Two are permanently resident and may be modified only through a system generation or reassembly. The remaining two are dynamically created and destroyed over time. The System Context Block, or SCB is permanently resident and contains parameters describing the overall status of the operating system and will not be described in further detail. There is one Device Context Block, or DCB, for each device type supported by the system. These context blocks contain information describing the device type and the status of each unit of that type and are permanently resident. Associated with each task within the operating system is a Task Context Block, or TCB, that is dynamically created when the task is initiated and is destroyed when the task is complete or is otherwise terminated. Since all task-initiated input/output processes reference 'files' rather than physical devices, the temporary association between file names and physical devices or datasets is recorded in File Context Blocks, or FCB's. An FCB is created whenever an association between a file name and device or dataset is established and is destroyed when that association is ended.

The Task Context Block and File Context Block structure is depicted in Figure 2. The base of this structure is the permanently resident Task Context Block Table located to the left. The length of the table defines the maximum number of tasks permissible in the system. Each entry in the table is either a null value or a pointer to a Task Context Block. When a new task is initiated, the EXECUTIVE procedure

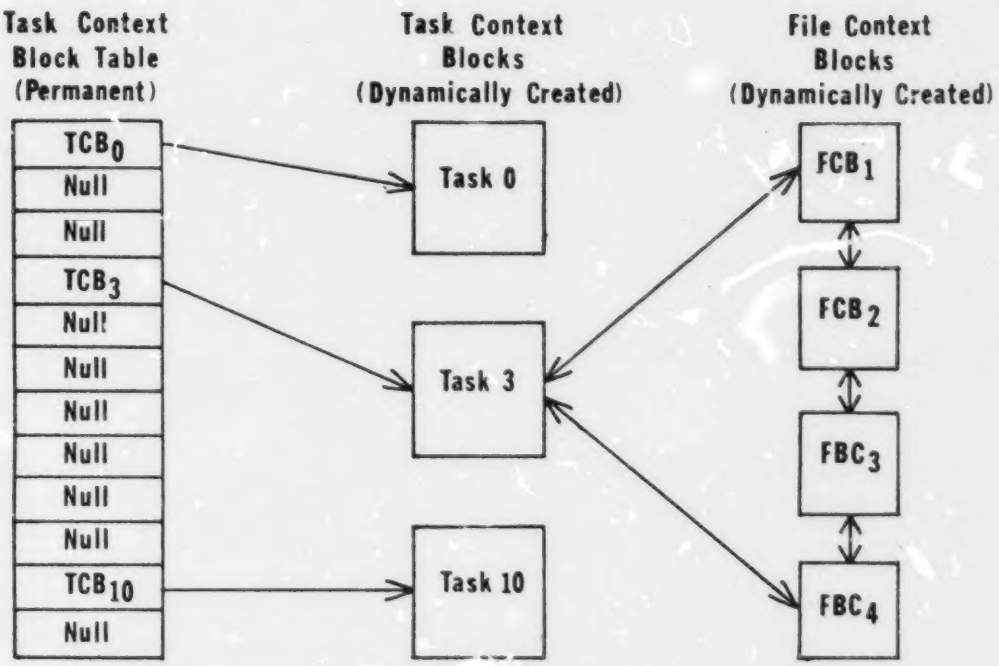

Figure 2. Task Context Block and File Context Block Structure

creates a Task Context Block and stores a pointer to the TCB in the first free position in the table. The relative position of the pointer in the table defines the 'task number' of the task which is used for internal identification only and is unknown to the human users of the system. The Task Context Block contains a set of descriptors fully defining the task and describing its current status. The TCB also contains two sets of list-processing pointers. The first is used to link the TCB into various scheduling queues, and the second links the TCB to any associated File Context Blocks that may have been created.

A File Context Block is created by the INPUT/OUTPUT MASTER CONTROL procedure whenever a task wishes to establish an association between a task-specified file name and a particular device or dataset. Since all task-initiated input/output processes are directed toward 'files,' the FCB contains the information required to translate file requests into requests to actual physical devices or datasets. As is shown on the right of Figure 2, the File Context Blocks of a task are linked in a bidirectional-linked list with the Task Context Block serving as the list head.

There are two types of File Context Blocks. The first associates a file name with a physical device; the second associates a file name with a dataset stored on a directory device. In the first case, the FCB contains the file name and a number specifying the device type and the unit number. In the second, the FCB contains the file name and the dataset name. In both types, there are entries with additional descriptors further defining the file and describing its current status.

The Device Context Block structure is shown in Figure 3. The Device Name Table at the left is the logical starting point of this structure and equates device names to type and unit numbers. All devices supported by the system are classed by type with individual

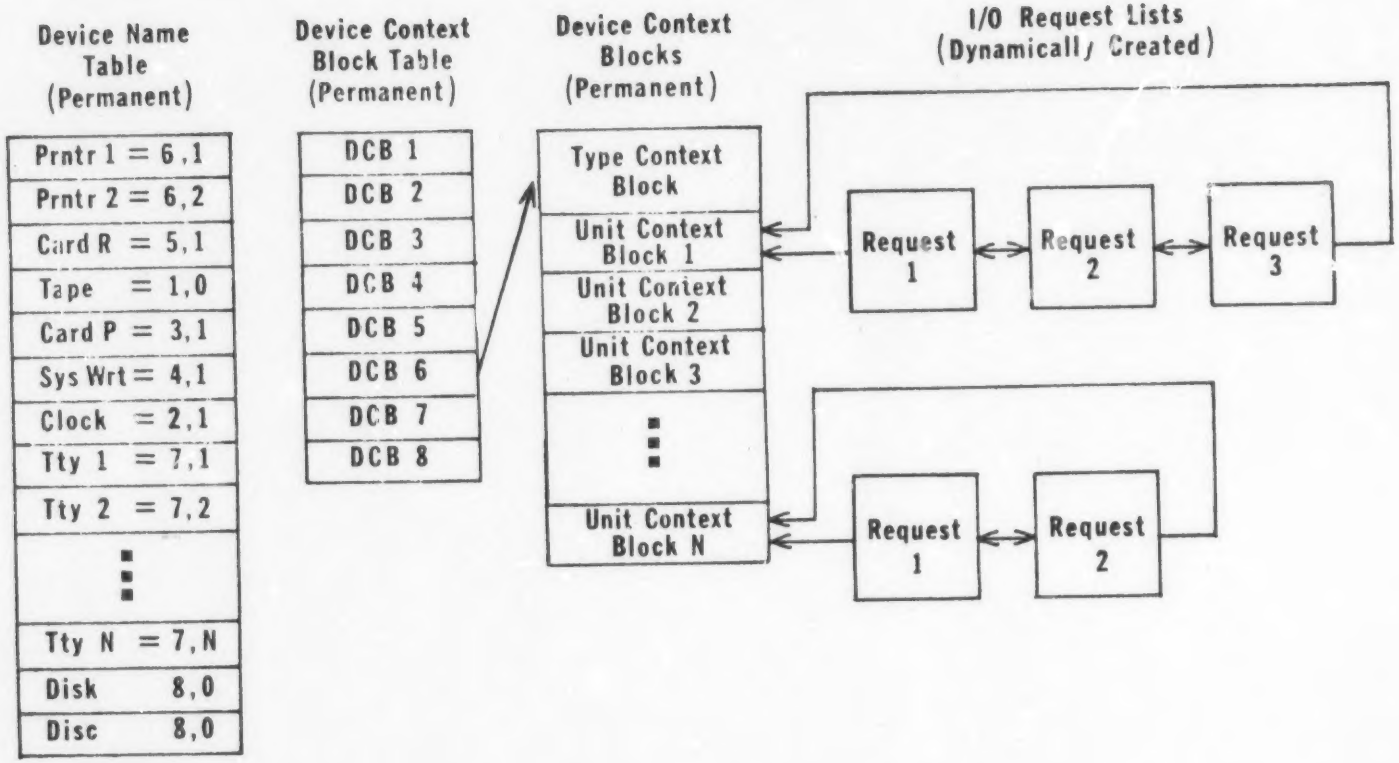

Figure 3. Device Context Block Structure

devices within a type referred to as units. When a task wishes to associate a file name with a particular device, the task provides the name of the device. A request to associate a file name with a dataset is assumed to refer to the device 'disk' unless it is otherwise specified. The INPUT/OUTPUT MASTER CONTROL procedure responsible for processing these requests locates the device name in the name table and retrieves the associated type and unit number. If the unit number is nonzero, the type and unit number is stored without modification in the newly created File Context Block. If the unit number is zero, this indicates that the unit is to be allocated by the operating system from among the available units of that type. The IOMC procedure selects an available unit and stores the type number and the unit number of the selected unit in the new FCB. All subsequent requests by the task specify the file name, and the Device Name Table is not accessed again until the task wishes to initiate a new input/output process.

Once the association between the file name and the device has been established, the logical starting point for all subsequent requests is the Device Context Block Table, shown second from the left in Figure 3. This table contains the addresses of the Device Context Blocks and is ordered by device type. Each device type has an associated Device Context Block and a Device Control Procedure (or I/O driver). The device type code is used as an index into the Device Context Block Table to the entry containing the address of the appropriate Device Context Block.

The Device Context Block consists of a Type Context Block defining the device type followed by one or more Unit Context Blocks, each describing the current status of a unit of that type. The Type Context Block is in a standard format common to all DCB's and contains

approximately 30 descriptors fully defining the device type. The Type Context Block also specifies the number of physical units of that type, the size of each Unit Context Block and the address of the Device Control Procedure controlling the device type.

The Unit Context Blocks that follow the Type Context Block are of variable length depending upon the device type although the first part of each Unit Context Block is in a standard format common to all types. Whereas the Type Context Block contains information on all devices of one type and is never altered, the Unit Context Block contains information on the current status of a particular unit and portions may be used as working storage by the controlling Device Control Procedure.

The Unit Context Block also contains a set of list-processing pointers which serve as the head of the request list for that unit. This list, shown on the right of Figure 3, is a bidirectional-linked list of outstanding requests against the particular unit. Input/output requests issued by a task are decoded by the INPUT/OUTPUT MASTER CONTROL Procedure into requests to an actual physical device. These requests are added to the end of the request list of the appropriate Unit Context Block, and the controlling Device Control Procedure is notified that a request is pending.

Scheduling and Memory Management. Any procedure within the system can request the EXECUTIVE to create a new task with a set of characteristics called a 'task profile.' By controlling the type and mix of the tasks that are created, procedures other than the EXECUTIVE can perform high-level or 'course' scheduling. However, once created, the low-level or 'fine' scheduling of the task is under the exclusive control of the EXECUTIVE procedure.

The EXECUTIVE maintains multiple queues of tasks that are ready to execute, one queue for each priority level. The number of priority levels (and queues) is installation dependent and can be readily modified through a system generation or reassembly. Higher priority queues are normally serviced before lower priority queues so that a task gains access to the processor only if no higher priority tasks are ready to execute. To ensure that extremely active higher priority queues do not lock out tasks on lower priority queues, each queue is assigned a 'queue ratio' which specifies the frequency with which a queue must be serviced even though higher priority tasks are ready to execute.

As part of its task profile, each task has an assigned priority range that defines the highest and the lowest priorities that the task can assume during its lifetime. A task gaining access to the processor for the first time is placed in the queue associated with its highest permissible priority. The task profile also specifies an initial time-quantum class which defines the maximum time period that a task will be permitted to continuously execute.

Once the task gains access to the processor, execution normally continues until the task requests to be suspended on a time-delay, issues an input/output request or attempts to exceed the established time-quantum. This latter case is termed a 'quantum overflow.' If an overflow occurs, the task is dropped to the next lowest priority level and is assigned to the next time-quantum class with a longer quantum, unless it is already on its lowest permissible level. If the task is already on its lowest permissible priority level, the task remains on that level and retains its current quantum class regardless of the fact that a quantum overflow has occurred. If the task executes for less than its current time-quantum, it is eligible to be moved to a higher priority level if the actual execution time falls within a shorter time-quantum class and if the task is not already on its highest

permissible level. If it is eligible to be moved upward, the task is assigned to the next shortest quantum class and is moved one priority level higher.

The task profile also assigns each task to one of three 'preemption classes.' A ready-to-run task in the 'immediate preemption' class will immediately preempt a currently executing task of lower priority. A task in the 'delayed preemption' class will immediately preempt lower priority memory resident tasks and swapping tasks that have been in memory for more than one time quantum. However, if the task to be preempted was swapped-in immediately prior to its execution, preemption will be delayed if the executing task has not been allowed to run for a specified period of time. The intent is to avoid unnecessary swapping while still ensuring that the higher priority task receives service within a short period of time. A task in the 'no preemption' class is not permitted to preempt a lower priority task and will only gain access to the processor when the lower priority task is removed.

Memory space for tasks is allocated and deallocated dynamically by the memory management subprocedure within the EXECUTIVE. Fixed-size partitions are not used, and through system calls, tasks can acquire and release memory as required. The task profile assigns each task to a 'memory size class' which specifies the maximum amount of memory the task can acquire. All language processors, editors, and utilities begin executing with a minimum of memory and expand and contract in size throughout a run. As the memory size of a task varies and crosses certain threshold points, it is raised or lowered in priority within the constraints of the maximum-minimum priority range specified in the task profile. A task's CPU use and memory size are additive in their effect upon priority so that, if permitted, a large, compute-bound

task will drift to a lower level than either a smaller compute-bound task or a larger I/O-bound task.

The task profile also assigns each task to one of four 'memory residency classes.' Tasks in classes one through three are eligible for swapping with each successive class having a higher 'sticking priority.' The sticking priority determines the order in which tasks are to be swapped with tasks having a lower sticking priority eligible for swapping before tasks with a higher sticking priority. Class four tasks are permanently resident and are never swapped.

When a task is ready to run, the scheduler first determines if the task currently resides in memory. If it does not, it must be swapped-in, and the memory management subprocedure is invoked. The subprocedure first scans for a block of free space large enough to accommodate the task, selecting the smallest if more than one of adequate size is available. If no block can be found, a calculation is made to determine if 'shuffling' or moving tasks within memory would create a single block of sufficient size. This calculation is not straightforward since there may be blocks of allocated space that cannot be moved until some ongoing input/output transfer, usually swapping, is complete. If shuffling is possible, a block of free space is created, and the task to be run is swapped-in. If it is not possible, one or more tasks must be swapped-out. The tasks currently in memory are scanned by memory class beginning with class one, the class with the lowest 'sticking priority.' Within a class, the scanning is by priority with the lowest priority task most eligible for swapping. However, a task of the same or higher priority that has been swapped-in preparatory to execution is not eligible to be swapped-out. This and a number of other considerations can greatly complicate the selection process. Once the one or more tasks

have been selected and swapped-out, some shuffling may be necessary before the task to be run can be swapped-in.

If the task to be swapped-in is a memory resident task, it will remain in memory throughout its existence. The shuffling algorithm will cause memory resident tasks and other tasks with high sticking priorities to drift downward in the memory address space so that memory very quickly becomes stratified by sticking priority. This speeds and simplifies the selection of tasks to be swapped-out and reduces shuffling since tasks with lower sticking priorities will tend to be adjacent at one end of the address space. Minimizing shuffling is important since shuffling can consume large amounts of CPU time, and if unrestricted, may be far less efficient than simple, brute force swapping.

This scheduler and memory management design permits the system to be tailored to a wide range of applications. By altering the queue ratios (the frequency with which lower priority queues are to be serviced even though higher priority tasks are ready to run) and the task profiles of the tasks created by the MONITOR and the BATCH procedures, the software can be dynamically reconfigured while the system is running. For example, if the application requires foreground time-sharing and memory resident, background batch, the MONITOR procedure would create time-sharing tasks with a low sticking priority, with a high priority range and with a relatively short initial quantum. The BATCH procedure would create batch tasks with a high sticking priority to force memory residency, with a low priority range and with a relatively long initial quantum. One or more batch streams might be supported depending upon operator specifications to the BATCH procedure. Within each type of task (i.e., time-sharing and batch), the

scheduler will dynamically allocate computational and memory resources based on task performance and will favor small, I/O-bound tasks over large, compute-bound tasks. However, the priority range concept ensures that the batch and time-sharing tasks do not overlap in priority unless this is felt to be desirable and is specified. Depending upon the queue ratios that are assigned, the batch stream(s) can be kept moving regardless of the degree of time-sharing activity, or they can be permitted to be locked out entirely during periods of extremely active time-sharing.

Clearly, a great many other scheduling configurations are possible. In a time-sharing service bureau environment, outside paying customers could be placed in a higher priority range than in-house software developers. In a process-control application, background software development could occur while operational process control tasks are active. Those process control tasks responsible for emergency or alarm conditions would be restricted to the highest priority level (where the priority range consists of one level only), placed in the immediate preemption class and assigned the highest sticking priority. Depending upon requirements, other process control tasks could be stratified on one or more lower levels, each with a unique task profile if necessary. Software development in a time-sharing or batch mode would be restricted to the lowest priority levels with a low sticking priority.

Conclusion

ETSS is an operational demonstration that a multilanguage, general-purpose operating system supporting both conversational time-sharing and multiprogrammed batch is possible on a medium-scale

computer. As equipment costs continue to decline, medium-scale systems such as ETSS should see increasingly wide use in the many application areas in which large-computer capabilities are not required. Although total hardware cost for the research and development prototype constructed four years ago is in excess of $200,000, a forty time-sharing port version of the system could be constructed today for $50,000 - $125,000 depending upon peripherals and disk storage requirements. This price range places medium-scale systems of this type well within the financial reach of many smaller educational, business, and research and development organizations whose computing requirements are extensive but not of a type or magnitude to economically justify a dedicated large-scale system.

BIBLIOGRAPHY

Fitzhugh, R. J. LRDC experimental time-sharing system reference manual, Volumes 1-3. Pittsburgh: Learning Research and Development Center, University of Pittsburgh, 1970.

Fitzhugh, R. J., and Pethia, R. D. Disk File Management in a Medium-Scale Time-Sharing System. Paper presented at the meeting of the Digital Equipment Corporation Users Society Fall Symposium, San Francisco, November, 1973.

Fitzhugh, R. J. Laboratory control with a medium-scale time-sharing system. Behavior Research Methods and Instrumentation, 1974, in press.